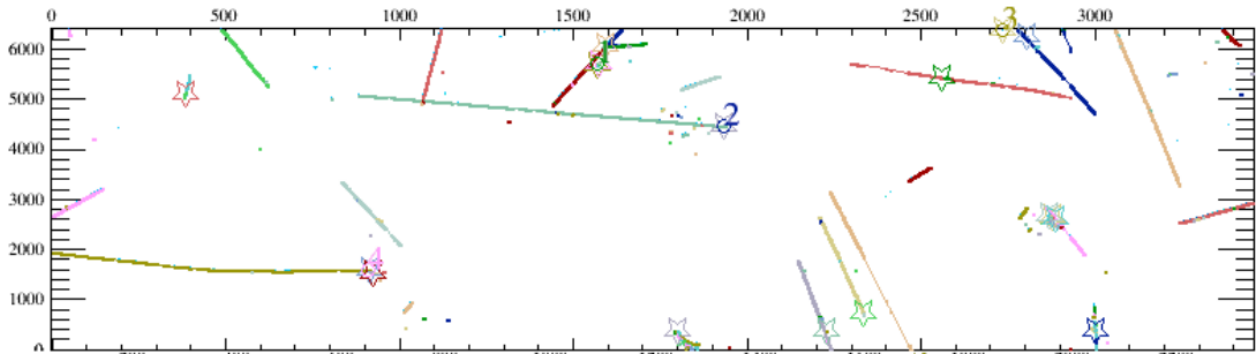

TrajCluster: A 2D Cluster Finder

Bruce Baller
Fermi National Accelerator Laboratory

April 28, 2016

Abstract

The TrajCluster module utilizes many features of the LineCluster module while rectifying its deficiencies. Both modules use a tracking strategy analogous to a Kalman Filter to reconstruct 2D line-like clusters of hits in a LArTPC wire plane. The LineCluster module, which uses the ClusterCrawlerAlg algorithm, constructs clusters by adding hits on adjacent wires to the leading edge of the cluster under construction. The tracking efficiency suffers by imposing a requirement that there be no more than one hit on a wire. Hits that were not properly reconstructed may be merged into a single hit using local information such as the cluster angle. As a result of this process, LineCluster produces another hit collection that must be written to the event.

The technique used in TrajCluster is to associate an arbitrary number of hits to a new trajectory point based solely on their proximity to the projected trajectory position. There is no requirement for these hits to reside on the same wire nor is there a requirement that they are associated uniquely with one trajectory. The subset of these hits that are used to define the position and charge of the new trajectory point is then determined and used to update the trajectory fit. Hits that are used in a trajectory fit are uniquely associated with that trajectory however. Hits that were not used remain in place and can easily be swapped between trajectories.

Contents

1 Introduction

A starting trajectory point, TP, is formed from two nearby hits; usually on two adjacent wires. The starting trajectory point is defined by the position of the first hit. The trajectory point direction is a unit vector from the first hit to the second. The standard coordinate system used by recob::Hit is (wire, tick), where wire is the wire number and tick is (roughly) the TDC tick at which the peak wire amplitude occurs. The tick value is translated into dimensionless "Wire Spacing Equivalent" units where the "time" = tick $\times$ v_{drift} $\times$ TDC sample rate / wire pitch. The electron drift velocity, v_{drift}, TDC sample rate and the wire pitch are obtained from the detector properties and LAr properties services. The trajectory is extrapolated by a distance S $\approx$ 1, the step size, and a search is made for nearby hits. New hits are added to a new trajectory point that is appended to the trajectory. The leading edge of the trajectory position and direction is then updated by fitting a variable number of trajectory points at the leading edge. The decision to use hits in the trajectory fit uses a "charge pull" as well as the impact parameter, Delta, between the projected trajectory and the hit. The Delta rms, the average charge and the charge rms of previously added trajectory points is recalculated at each step and is used to make decisions on hits found on subsequent steps. This process continues until no new hits are found.

The fitting method is based on the assumption that reconstructable particles have a minimum range of 2 - 3 wires, corresponding to a kinetic energy $\gtrsim$10 MeV for muons and pions. The effects of multiple Coulomb scattering can be largely ignored over this length scale since the error will be dominated by the hit position error. As a result, reasonable results can be obtained by fitting a variable number of point to a line.

After all trajectories are made in a wire plane, a search is made for 2D vertices in that plane. After trajectories are made in all wire planes, the 2D vertices in each plane are matched to create 3D vertices. An additional step for 3-plane LArTPCs is to split trajectories in the third plane where a 2D vertex should exist given the knowledge of 2D vertices in the other two planes. The most common reason is that there are two particles which are traveling away from each other without any detectable kink in the third plane view. Trajectories in the third plane with this "hammer" topology are then split and associated with a new 2D vertex and the 3D vertex.

Two features of the procedure described above are evident. One can start at the DownStream (DS) end of the hit collection, defined to be the larger wire number, and step to the UpStream (US) end, those with smaller wire number, of the hit collection with the implicit assumption that the wire number increases in the neutrino beam direction. Alternatively, one can start at the other end. It is not obvious that stepping in any one direction will be better than the other for every experiment. TrajCluster provides the option of stepping in either direction via the fcl setting, Mode. Stepping is done from US $\rightarrow$ DS when Mode = 1 and from DS $\rightarrow$ US when Mode = -1. An additional option, Mode > 1 is not functional yet.

A second feature is that high quality information is only available after some number of trajectory points have been added to the trajectory. Poor decisions may be made early in trajectory construction. Two avenues are available to mitigate this problem. One is to use reverse propagation, a concept from the Kalman filter technique, where after constructing a trajectory, the procedure is reversed to improve the trajectory at the beginning. A second approach is to use human judgment to estimate the initial conditions and only gradually rely on trajectory-specific information. The second approach has been found to be adequate and is described in section 5.

The data products produced by this module are:

- std::vector<recob::Cluster>
- std::vector<recob::Vertex>

- std::vector<recob::EndPoint2D>
- art::Assns<recob::Cluster, recob::Hit>
- art::Assns<recob::Cluster, recob::Vertex, "Vtx Indices"> where Vtx Indices point to a 3D vertex at each end
- art::Assns<recob::PFParticle>
- art::Assns<recob::Cluster, recob::PFParticle>

2 Algorithm flow

Processing of an event begins when TrajCluster_module passes the art event handle to RunTrajClusterAlg. Reconstructed hits are stored in the fHits vector that resides in the TjStuff struct. This struct contains all data objects that need to be passed to the utility routines in TCAlg/Utils. These objects are:

- std::vector<art::Ptr<recob:Hit>> fHits - the event hit collection.
- std::vector<Trajectory> allTraj - All trajectories reconstructed in all planes.
- std::vector<short> inTraj - ID of the trajectory that uses this hit.
- std::vector<short> inClus - Index of the cluster that uses this hit. Passed to TrajCluster_module.
- std::vector<ClusterStore> tcl - All clusters reconstructed in all planes. Passed to TrajCluster_module.
- std::vector<VtxStore> vtx - All 2D vertices found in all planes. Passed to TrajCluster_module.
- std::vector<Vtx3Store> vtx - All 3D vertices found in the TPC. Passed to TrajCluster_module.
- float fUnitsPerTick - Conversion factor from TDC ticks to Wire Spacing Equivalent units.
- std::vector<std::vector<Trajectory>> trial - All trajectories in all planes for each trial run.
- std::vector<std::vector<VtxStore>> inTrialVtx - All 2D vertices found in all planes for each trial run.
- std::vector<std::vector<Vtx3Store>> inTrialVtx3 - All 3D vertices found in the TPC for each trial run
- std::vector<TjPairHitShare> tjphs - List of trajectory pairs that share hits.

The inTraj vector has the same size as fHits and provides the association from hit → trajectory. A hit with index iht that is not used in a trajectory has inTraj[iht] set to 0. An association to a trajectory is made by setting inTraj[iht] to the ID of the trajectory. A positive non-zero value of the trajectory ID indicates that it is stored in the vector all trajectories, allTraj. The trajectory under construction, the "work" trajectory, has an a ID of -1. A hit associated with the work trajectory therefore has inTraj[iht] = -1. Another index association exists from trajectory → hit to obviate the need for repeated searches in the large hit collection. Frequent checks ensure that the two associations are consistent.

Processing is done in one or more "trials", where a trial is the reconstruction of all trajectories in the TPC using a set of reconstruction settings. The only reconstruction setting that is currently implemented is Mode = ±1 as specified in the fcl file.

The main processing loops in RunTrajClusterAlg are over all planes in all TPCs in all cryostats. Two preparatory measures are taken to reduce unnecessary computation. A unique "CTP" integer code is defined for each plane in the experiment. Each trajectory, trajectory point and 2D vertex has a CTP assignment to simplify searching for compatible objects. The second measure is to define the index of the first hit and the last hit on each wire in the current plane, significantly narrowing searches in the hit collection. This is done in the GetHitRange algorithm, which takes

advantage of the fact that hits are naturally ordered by plane, wire and tick when they are created in the hit finder module. A consistency check is made to ensure that this ordering is indeed correct. GetHitRange defines the WireHitRange vector of std::pairs where the the first element of the pair is the index of the first hit on the wire and the second element is the end of the last hit on the wire. A WireHitRange value of -2 means that there are no hits on the wire and a value of -1 means that the wire is dead.

The ReconstructAllTraj algorithm next reconstructs all trajectories in the current plane and stores them in the allTraj vector. A search is then made for 3D vertices. If there is more than one trial, all vectors for the current trial are stored. After all trials have been completed the AnalyzeTrials and AdjudicateTrials algorithms are called to determine the best set of trajectories to use from all trials. The best set of trajectories is put into the allTraj vector. Significant development is required before the multiple trials feature is useful.

An optional call to the TagAllTraj algorithm is made to classify trajectories as track-like or shower-like. This algorithm defines the PFParticle hierarchy. Additional development is needed to quantify the quality of PFParticles produced by this algorithm.

The last steps are to convert all trajectories into clusters in the MakeAllTrajClusters algorithm and to check the hit $\leftrightarrow$ cluster associations. The produced data products are passed to TrajCluster_module via the get methods in the TrajClusterAlg header file.

2.1 Data structs

The structs that define a trajectory and a trajectory point are located in TCAlg/DataStructs.h and are shown in Figure 1. Each trajectory point, or "TP", consists of the charge weighted position of all hits that are "used" to define its position, HitPos. The vector Hits includes all that are close to the projected position as determined by AddHits. The vector UseHit specifies which hits will define the trajectory position and charge. The values of the UseHit bool vector are initially specified in FindUseHits using the history of previously added hits to decide which hits are appropriate to use in the TP under construction.

It should be noted that the sizes of the Hits and UseHit vectors are not changed after they are defined in AddHits. A hit may be associated with several trajectories in the sense that the hit index appears in the Hits vectors of trajectory points in those trajectories, but the UseHit bool may only be set true for the hit in one of the points.

Each trajectory point has a charge (Chg) which is the sum of the Integral() of all used hits, the average charge (AveChg) of the previous NPtsAve points, charge pull (ChgPull), and a hit position (HitPos). The impact parameter, Delta, of the projected trajectory at HitPos is stored before the trajectory fit is done. This convention ensures that Delta is not biased by it's presence in the fit. The error^2 of all hits used to define the position, HitPosErr2, is specified as described in section 6.

A linear fit including the just added TP is done using NTPsFit trajectory points at the leading edge of the trajectory. When the fit is found to be acceptable as described in section 5, the parameters FitChi, NTPsFit, ChgRMS, the trajectory point angle Ang, and the angle error, AngErr, are defined.

A trajectory is composed of a vector of TPs, pointers to vertices, and a number of useful variables. The EndPt array of size two gives the index of the first and last points on the trajectory that have charge. These end points are the maximum physical extent of the trajectory. Trajectory points that are outside the EndPt boundaries were masked off and likely are used in a different trajectory. The Vtx array of size two gives the index of a 2D vertex which may be attached to either end. The average charge and charge rms of all points on the trajectory is stored here. This information is useful when deciding whether two trajectories should be merged into one. The truth

variables, TruPDG, TruKE and EffPur are filled when the MC particle matching code is used. The AlgMod bitset is described in section 9.

```
struct TrajPoint {
  CTP_t CTP {0};                    ///< Cryostat, TPC, Plane code
  std::array<float, 2> HitPos {{0,0}}; // Charge weighted position of hits in wire equivalent units
  std::array<float, 2> Pos {{0,0}}; // Trajectory position in wire equivalent units
  std::array<float, 2> Dir {{0,0}}; // Direction
  float HitPosErr2 {0};          // Uncertainty^2 of the hit position perpendiclar to the direction
  // HitPosErr2 < 0 = HitPos not defined because no hits used
  float Ang {0};                 // Trajectory angle (-pi, +pi)
  float AngErr {0.1};              // Trajectory angle error
  float KinkAng {-1};             // Just what it says
  float Chg {0};                 // Charge
  float AveChg {-1};              // Average charge of last ~20 TPs
  float ChgPull {0.1};           //  = (Chg - AveChg) / ChgRMS
  float Delta {0};               // Deviation between trajectory and hits (WSE)
  float DeltaRMS {0};            // RMS of Deviation between trajectory and hits (WSE)
  unsigned short NTPsFit {2}; // Number of trajectory points fitted to make this point
  unsigned short Step {0};       // Step number at which this TP was created
  float FitChi {0};              // Chi/DOF of the fit
  std::vector<unsigned int> Hits; // vector of fHits indices
  std::vector<bool> UseHit; // set true if the hit is used in the fit
};

// Global information for the trajectory
struct Trajectory {
  short ID;
  CTP_t CTP {0};                      ///< Cryostat, TPC, Plane code
  unsigned short Pass {0};            ///< the pass on which it was created
  short StepDir {0};                 /// -1 = going US (CC proper order), 1 = going DS
  unsigned short ClusterIndex {USHRT_MAX};   ///< Index not the ID...
  std::bitset<32> AlgMod;         ///< Bit set if algorithm AlgBit_t modifed the trajectory
  unsigned short PDG {0};            ///< shower-like or track-like {default is track-like}
  unsigned short ParentTraj {USHRT_MAX};     ///< index of the parent (if PDG = 12)
  float AveChg {0};                   ///< Calculated using ALL hits
  float ChgRMS {1};                 /// Normalized RMS using ALL hits. Assume it is 100% to start
  int TruPDG {0};                    ///< MC truth
  int TruKE {0};                     ///< MeV
  float EffPur {0};                    ///< Efficiency * Purity
  std::array<short, 2> Vtx {{-1,-1}};      ///< Index of 2D vertex
  std::array<unsigned short, 2> EndPt {{0,0}}; ///< First and last point in the trajectory that has a hit
  std::vector<TrajPoint> Pts;    ///< Trajectory points
  std::vector<TrajPoint> EndTP {(2)} ;    ///< Trajectory point at each end for merging
};
```

Figure 1: Trajectory and trajectory point structs.

3 ReconstructAllTraj

This algorithm reconstructs all trajectories in the current plane, finds 2D vertices and stores the results in the TjStuff struct. Several passes through the hit collection can be specified by the user via fcl settings. The settings for each pass are a) whether to reconstruct large angle trajectories, (LAStep), b) the minimum number of TPs that should be allowed in the trajectory fit (MinPtsFit), c) the minimum number of trajectory points (MinPts), and d) the maximum trajectory - 3D vertex separation distance (MaxVertexTrajSep). Experience has shown that allowing the reconstruction of less common large angle trajectories to soon can impair reconstruction of more common small angle trajectories. The fcl setting LargeAngle defines a rough boundary between normal stepping and large angle stepping. The convention used is that a track traveling along the drift direction has an angle of 90°. The multiplicity of hits per wire on tracks exceeding $\approx 70°$ increases dramatically. Below this angle the hit multiplicity per wire is usually 1 - 2.

The algorithm flow in ReconstructAllTraj is to loop over all user-specified passes, then over each hit, iht, on each wire, iwire, in the current plane. Hits, jht, on the adjacent wire, jwire, are tested to see if they could lie on the same trajectory. The test, done in TrajHitsOK, returns true if the StartTick() of iht(jht) and the EndTick() of jht(iht) overlap in time as expected.

When compatible hits are found the work trajectory is re-constructed by StartWork and tailored

for the current plane, pass and stepping direction. A "bare" trajectory point is made and pushed onto the work Pts vector. A bare trajectory has a position, direction and angle but no hits. The work trajectory, consisting of the single bare trajectory point, is passed to AddHits which adds hits to the trajectory point, decides which of those hits to use and defines the trajectory point HitPos. This process ensures that all hits that may be a part of a hit multiplet of which iht is a member are included in the trajectory point. See section 7 for further details.

Further construction of the trajectory is done in StepCrawl as described in the next section. A possible result is that the work trajectory is high quality but fails the cuts for the current pass. If this occurs, the flag kTryWithNextPass is set true, triggering ReconstructAllTraj to call StepCrawl again with the next pass cuts. The work trajectory is abandoned if the second attempt is unsuccessful.

The quality of the successfully constructed work trajectory is assessed in CheckWork. Algorithms within CheckWork may modify the work trajectory to improve it's quality. An option exists to reverse propagate the work trajectory in ReversePropagate.

The work trajectory is then pushed onto allTraj by StoreWork. A search is made for 2D vertices after all trajectories in the current plane and current pass have been reconstructed.

After all passes have been completed, the algorithm FindJunkTraj constructs trajectories from unused hits. See section 8 for details.

4 StepCrawl

StepCrawl is the primary algorithm for completing construction of the work trajectory. The position and direction of the leading edge of the trajectory are defined by the last trajectory point on the Pts vector. A local bare trajectory point, ltp, is constructed from the last trajectory point. This point is moved through the (wire,time) space while stepping.

A step variable is incremented for every iteration. The step size is chosen using the LargeAngle variable. Most trajectories are small angle so stepping by one wire, 1/ltp.Dir[0], is the most efficient. The step size is set to one for large angle trajectories. The local TP is moved by this amount and the position and direction transferred to a copy of the last trajectory point on the work trajectory. This new point, tp, is pushed onto the work trajectory and a call made to AddHits. If no hits were added to the point, it is removed from the trajectory and the number of missed steps without a signal is counted. Stepping halts if the number of missed steps without a signal exceeds the user cut, MaxWireSkipNoSignal. If hits were added to the point but none were selected to be used, the point is left on the trajectory and a count made of the number of missed steps with a signal. Stepping halts if the number exceeds the user cut MaxWireSkipWithSignal.

The just-added point, tp, now has hits associated with it, some of which have been used to define the hit position. The trajectory at this new point is defined by UpdateTraj. The position and direction are copied to ltp. It is possible that the last trajectory point was masked off, i.e. UseHit set false for all hits, because of a poor fit. If this happens repeatedly, it is an indication that the trajectory curvature is too large to follow using the current pass settings. The trajectory is inspected for a kink in the GottaKink algorithm. The last three points that were added to the work trajectory are fit to a mini-trajectory. The angle between this mini-trajectory and the presumed kink point which is the fourth point from the end is calculated. If this angle difference exceeds the user selected cut, KinkAngCut, or if the significance of the angle difference is very large, the last 3 points are masked off and stepping is halted.

The process of masking off trajectory points is done by the utility algorithm MaskTrajEndPoints. The steps are to set UseHit false for all hits on these points, and set inTraj to zero for these

hits. Another utility algorithm SetEndPoints is called whenever points are masked off.

Stepping continues until the number of missed steps exceeds the user cuts, a kink is found, the trajectory leaves the TPC volume or the maximum number of steps is reached.

5 UpdateTraj

UpdateTraj updates the last trajectory point, lastTP, of the trajectory that is passed to it. The first steps are to set lastTP.Delta using the PintTrajDOCA utility routine and update the lastTP charge, average charge, charge rms and the charge pull.

Next a decision is made on the number of trajectory points to fit. The decision is simple when there are less than 3 points; the points are all fit, the fit parameters of the point are updated and control is returned to the calling algorithm.

In general, the approach is to fit as many points as possible at the leading edge of the trajectory to a straight line while keeping the χ^2/DOF less than 2. This ensures that the projection error is minimized, enabling the capability to track cosmic ray muons through δ-ray showers. The default action is to increment the number of points fit, NPtsFit, of the previous point by one. A more complex situation arises when the fit χ^2/DOF of the previous point exceeds 2. This is an indicator of poorly chosen hits, an onset of a kink or a wandering trajectory. A large χ^2/DOF may also occur when reconstructing somewhat curved trajectories after stepping through a large section of dead wires. This is obvious because there are no points in the dead region to update the trajectory.

The algorithm attempts to distinguish between these situations by calculating the rate of increase of χ^2/DOF in the last few points. A sudden increase in χ^2/DOF indicates the onset of a kink. In this situation, the last point is masked off, i.e. not used in the trajectory fit. The position of the last point is updated and control returned to the calling routine. If the increase in χ^2/DOF is gradual, it is likely that the trajectory is curved, in which case the number of fitted points is reduced and the trajectory re-fit until χ^2/DOF is less than 2.

6 FitTraj

The supplied trajectory is fitted to a line with an origin at originPt with the number of fitted trajectory points npts. The fit direction, fitDir, specifies whether points on either side of the originPt or points straddling originPt should be fit. The output of this algorithm is a bare trajectory with a defined position, direction and FitChi. A mnemonic representation of the interpretation of the input parameters is presented in the comments. Overloaded versions of this algorithm are provided to fit the leading edge points of a supplied trajectory to a line.

The error ellipse of a trajectory point consisting of a single hit is shown schematically in figure 2. The error in the wire dimension is simply $1/\sqrt{12}$. The error in the time dimension is taken to be a user-defined fraction, HitErrFac, of the hit RMS(), which is then converted to WSE units by UnitsPerTick. The determination of HitErrFac can be done by checking the fit χ^2/DOF for long straight small angle trajectories where the wire error is negligible. HitErrFac should be adjusted until χ^2/DOF ≈ 1. The time error for single hit and multi-hit trajectory points is calculated in HitsTimeErr2.

The fit begins by rotating the selected trajectory point hit positions, HitPos, into a coordinate system oriented along the originPt direction. The weight originally assigned to each point is the hit position error^2, HitPosErr2, which is the position error transverse to the trajectory direction. The weight is multiplied by the charge pull of the trajectory point to reduce the sensitivity of the fit to high charge hits.

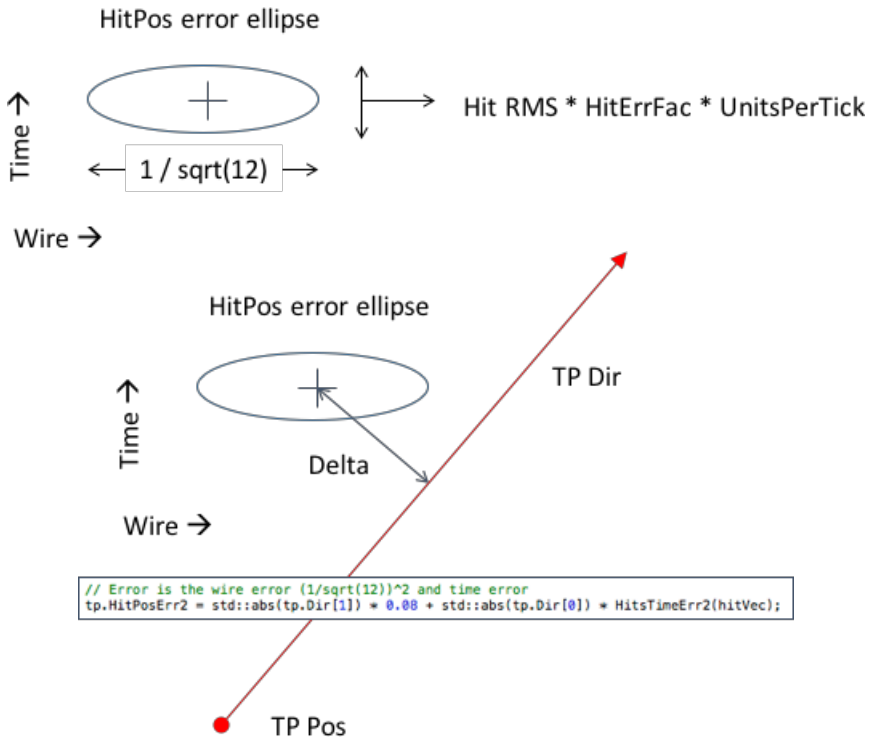

Figure 2: Top: Error ellipse in WSE units. HitErrFac is a fcl parameter (≈0.4). The hit rms in ticks is converted to WSE units by the scale factor UnitsPerTick. Bottom: Definition of Delta, the impact parameter between a trajectory point and the HitPos of a trajectory point. The calculation of HitPosErr2 is shown in the inset.

After the fit is done, the trajectory is rotated back to the (wire, time) coordinate system. The direction cosines are calculated and the position translated to the position of originPt. The fit χ^2/DOF is calculated if there are more than three points.

7 Hit multiplets

A hit finder module searches for wire signals that are above some threshold. The wire signal is fit to a variable number of Gaussian distributions that are meant to represent energy depositions from particles in the TPC. As mentioned above, this assumption breaks down for large angle tracks. Ionization fluctuations on such tracks typically result in the reconstruction of many hits that do not always represent the full energy deposition at the location of the hits. Such hits are fitted as a multiplet of overlapping Gaussian distributions. The quality of individual hits in the multiplet is low but the proximity of these hits to each to each other indicates that they are produced by the same process; the passage of a large angle track.

Given the uncertainty of defining a high quality hit in a multiplet, TrajClusterAlg requires the specification of a minimum hit separation significance, MultHitSep. To motivate this requirement, consider the case where the hit finder has reconstructs two hits in one multiplet, i.e. two overlapping Gaussian fits. In this situation, TrajClusterAlg may decide that the two hits should be considered as a single entity because of their separation or as separate objects. The importance is clear because TrajCluster algorithms use charge to guide reconstruction.

Two utility algorithms provide a means of handling internally defined multiplets. When passed

the index of a hit, theHit, GetHitMultiplet returns a vector all hits that are within MultHitSep of each other with the requirement that theHit be a member of the multiplet. When passed the index of a hit, theHit, HitMultipletPosition returns the charge-weighted tick, rms and charge of the hit multiplet of which theHit is a member.

8 FindJunkTraj

Failures of the stepping algorithm will inevitably occur, in particular for tracks whose charge is deposited on only a few wires in one view. FindJunkTraj reconstructs crude trajectories by collecting unused hits that are within a user-specified distance, JTMaxHitSep, of each other. The collected hits are stored in a flat vector, tHits, and a check made to see if they a "ghost" of an existing trajectory. If so, the hits are assigned to the existing trajectory. If not, the hits are pass to MakeJunkTraj.

The hits are fitted to a line to get a rough estimate of the orientation, sorted along that axis and then placed in a series of trajectory points that are separated by 1 WSE unit. No attempt is made to fit the trajectory. The trajectory point positions are simply set to the hit positions.

9 Development tools

9.1 Reports

A copious amount of debugging information will be generated when DebugPlane, DebugWire, and DebugHit are set ≥ 0.

To view a stepping report showing the construction of all points, DebugPlane and DebugWire should be set to the plane/wire of the first hit on the trajectory. DebugHit should set to the approximate position of the hit, ± 10 ticks. The bool variable prt is set true in ReconstructAllTraj when the hit is encountered.

A report of merging trajectories is produced when DebugPlane is set to the plane in which the report is desired. DebugWire should be set < 0, and DebugHit > 0. The bool variable mrgPrt is set true when the merging code is used.

A report of 2D vertex finding is made when DebugPlane is set to the plane in which the report is desired. DebugWire should be set > 0, and DebugHit < 0. The bool variable vtxPrt is set true when the vertex finding code is used.

A report of 3D vertex finding is made when vtxPrt is set true if DebugPlane > 0 and DebugHit $= 6666$.

A trajectory summary report is made when DebugPlane > 2. An example is shown in figure 3.

9.2 Algorithm usage

There are a fair number of algorithms that may modify a trajectory. The order in which these algorithms are used is hard-coded but the capability exists to turn off many of them. This may be useful to determine if some do more harm than good in a particular experiment. This is implemented as a std::bitset fUseAlg in TrajClusterAlg.h, a typedef enum in DataStructs.h and AlgBitNames in DataStructs.cxx. The algorithms ChainMerge and ReversePropagation may be turned off by setting the fcl input SkipAlgs: ["ChainMerge" , "RevProp"]. A list of all algorithm names is printed out if any of the SkipAlg entries is not recognized. The "Killed" bit is set when a trajectory is declared obsolete by a call to MakeTrajectoryObsolete.

```
****** 3D vertices ***************************************__2DVtx_Indx__*******
Vtx  Cstat  TPC Proc     X       Y       Z    XEr  YEr  ZEr  pln0 pln1 pln2  Wire
  0      0    0    1   195.4   -10.7   164.6  0.2  0.8  0.8 6666    0    1   -1    Matched in all planes
************ 2D vertices ************
Vtx   CTP    wire     error   tick     error  ChiDOF  NCl  topo  traj IDs
  0     1   576.5 +/-  2.0  4305.9 +/-  2.0     0.0    3     1       3_1   4_1   5_0
  1     2   548.3 +/-  2.0  4320.2 +/-  2.0     0.0    2     0       7_0   8_0
  2     0   643.0 +/-  2.0  4306.9 +/-  0.5     0.0    2     8       1_0
TRJ  ID CTP Pass Pts frm  to     W:Tick   Ang   AveQ     W:T      Ang   AveQ  ChgRMS Hits/TP __Vtx__ PDG Parent TRuPDG   EP   KE
TRJ   1  0    0  238   1 237   961:5377 -2.68    78   672:4400 -2.63    76   0.31    1.00   2  -1    0 65535    13  0.99  739 UseHiMultEndHits FillGap
TRJ   2  0    1    2   0   1   757:4677 -3.01   101   756:4676 -3.01   107   1.00    1.00  -1  -1    0 65535     0  0.00    0 757:4677 JunkTj
TRJ   3  1    1   87   1  86   577:4299  1.96   179   491:5380  2.29   225   0.35    1.70  -1   0    0 65535    13  0.85  739 Kink ManyHitsAdded
TRJ   4  1    0   14   1  13   521:5018  1.96   124   504:5233  2.27   113   0.60    1.00  -1   0    0 65535     0  0.00    0 UseHiMultEndHits
TRJ   5  1    0  192   1 191   770:3730  2.67   145   578:4302  2.74   104   0.25    0.98   0  -1    0 65535  2212  0.97  365 UseHiMultEndHits FillGap
TRJ   6  1    1    2   0   1   768:3733  2.56   291   767:3737  2.56   316   1.00    1.00  -1  -1    0 65535     0  0.00    0 768:3733 JunkTj
TRJ   7  2    0  160   1 159   707:3735  2.51   169   549:4315  2.19   129   0.20    0.94   1  -1    0 65535  2212  0.93  365 HiEndDelta UseHiMultEndHits
TRJ   8  2    0  231   1 230   781:5386 -2.51   107   550:4325 -2.61   114   0.25    1.00   1  -1    0 65535    13  0.99  739 UseHiMultEndHits FillGap
TRJ   9  2    1    5   1   4   636:4686 -2.99    78   633:4683 -2.95    78   1.00    0.80  -1  -1    0 65535     0  0.00    0
TRJ  10  2    1    9   1   8   706:3737  2.45   342   699:3767  2.47   315   0.24    0.89  -1  -1    0 65535     0  0.00    0
```

Figure 3: Trajectory summary report. The positions of 3D vertices is shown at the top, followed by the indices of the 2D vertices that are matched to it. The (wire, tick) positions and position errors of 2D vertices in all planes is printed next. The IDs of trajectories that are attached to the vertex are printed on the right with a suffix _0 or _1 that show which end of the trajectory the vertex is attached to. A summary of each trajectory is printed below.

```
TrajCluster algorithm counts
MaskHits              1098 Kink               3112 CWKink               20 CWStepChk          0
TryNextPass              5 RevProp               0 ChkHiMultHits      414 SplitTraj        310
Comp3DVx               430 HiEndDelta          603 Hammer2DVx       170 JunkTj         10178
Killed                   0 StopAtVtx            53 EndMerge            0 TrimHits         319
ChkHiMultEndHits         0 ChainMerge            0 FillGap           888 UseGhostHits    701
```

Figure 4: The number of times each algorithm modified (or created) a trajectory while reconstructing 800 MicroBooNE BNB-only neutrino interactions. The FindJunkTraj algorithm created 10178 trajectories.

When any of the algorithms modifies a trajectory, a bit is set in the AlgMod variable of the trajectory (figure 1). The algorithm names that modified the trajectory are printed out in the summary report (figure 3).

9.3 Stepping example

Figure 5 shows selected information from the detailed stepping report. In this run, the settings were Mode = 1, DebugPlane = 2, DebugWire = 471 and DebugHit = 4410 and the output directed to a log file. Each trajectory point is printed on a line prepended with the characters "TRP", the pass number and "+" or "-" which represents the stepping direction, StepDir. The trajectory point information was extracted from the log file by using the command "grep TRP1 debug.log ¿ trp.txt". The text file was then imported into Excel to produce the plots in the figure.

Most of the trajectory points have one hit associated with it and that hit was used in almost all points. The trajectory in (wire, tick) space is shown in the top figure. The middle figure shows the results from the trajectory fit. The number of points fitted to a line increases to 60 until a small angle scatter is encountered near wire 530. The fit χ^2/DOF rises gradually to 2 at this wire at which point UpdateTraj decreased the number of fitted points to 30 with a resultant drop in χ^2/DOF to 1. The pattern repeats again near wire 568 and more frequently towards the end of the trajectory where a kink is encountered. This is most likely a stopping particle (pion?) with a decay. This hypothesis is supported by trend of increasing charge as shown in in the bottom figure.

Figure 5: Top: Trajectory spanning the range from wire 470 to 613. Middle: The deviation, Delta, between the projected trajectory position at the wire and the trajectory point on that wire (blue dots). The fit χ^2/DOF at each trajectory point (orange dots) using the previous NTPsFIt trajectory points (grey dots). Bottom: Charge and average charge of each trajectory point.